Motivations

Suppose you will define classes to model circles, rectangles, and triangles. These classes have many common features. What is the best way to design these classes so to avoid redundancy? The answer is to use inheritance.

Introduction to Inheritance

Technique for deriving a new class from an existing class.

Existing class called **superclass, base class, or parent class.**

New class is called **subclass, derived class, or child class.**

Introduction to Inheritance

- Subclass and superclass are closely related
  - Subclass share fields and methods of superclass
  - Subclass can have more fields and methods
  - Implementations of a method in superclass and subclass can be different
  - An object of subclass is automatically an object of superclass, but not vice versa
    - The set of subclass objects is a subset of the set of superclass objects. (E.g. The set of `Managers` is a subset of the set of `Employees`.) This explains the term subclass and superclass.

Introduction to Inheritance

Why inheritance?

`Employee` class:

`name, salary, hireDay;`

`getName, raiseSalary(), getHireDay().`

`Manager` **is-a** `Employee`, has all above, and

Has a bonus

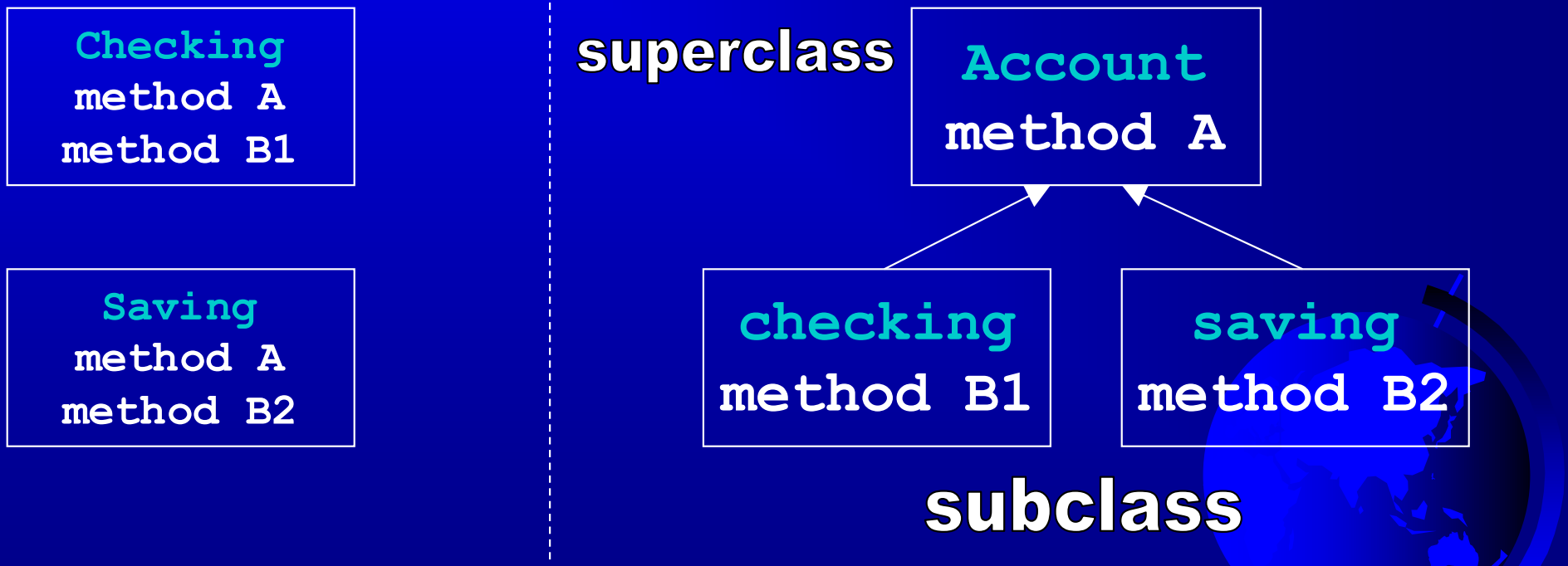
getsalary() computed differently

Instead of defining `Manager` class from scratch, one can derive it from the `Employee` class. Work saved.

Introduction to Inheritance

Why inheritance?

Inheritance allows one to factor out common functionality by moving it to a superclass, results in better program.



Deriving a class

```
class Employee
{
  public Employee(String n, double s, int year, int
  month, int day) {…}

  public String getName(){…}
  public double getSalary() {…}
  public Date getHireDay(){…}
  public void raiseSalary(double byPercent) {…}

  private String name;
  private double Salary;
  private Date hireDay;
}
```

Deriving a class

Extending `Employee` class to get `Manager` class

```
class Manager extends Employee
{ public Manager(…) {…}          // constructor

  public void getSalary(…) {…}  // refined
  method

  // additional methods
  public void setBonus(double b){…}

  // additional field
  private double bonus;
}
```

//ManagerTest.java

Fields of subclass

- Semantically: Fields of superclass + additional fields
  - `Employee`
    - `Name, salary, hireday`
  - `Manager`
    - `name, salary, hireday`
    - `bonus`
- Methods in subclass cannot access private fields of superclass.
  - After all, subclass is another class viewed from super class.

Fields of subclass

Static instance fields are inherited but not duplicated in subclass.

```
class Employee //StaticInherit.java
 {   public Employee (...)
     { ...
       numCreated++;
      }
     public static int getNumCreated()
     { return numCreated; }
     ...
     private static int numCreated=0;
 }
Manager b = new Manager(...);          // numCreated = 1
Employee e = new Employee(...);        // numCreated = 2
Employee.getNumCreated();              // 2
Manager.getNumCreated();               // 2
```

Fields of subclass

To count number of Managers separately, declare a new static variable in `Manager` class

```
class Manager extends Employee
{   public Manager (...)
     { ...
       numManager++; }
     public static int getNumCreated()
     { return numManager; }
    ...
     private static int numManager=0;
}
```

//StaticInherit.java

Are superclass's Constructor Inherited?

No. They are not inherited.

They are invoked explicitly or implicitly.

Explicitly using the super keyword.

A constructor is used to construct an instance of a class. Unlike properties and methods, a superclass's constructors are not inherited in the subclass. They can only be invoked from the subclasses' constructors, using the keyword super. *If the keyword super is not explicitly used, the superclass's no-arg constructor is automatically invoked.*

Superclass's Constructor Is Always Invoked

A constructor may invoke an overloaded constructor or its superclass's constructor. If none of them is invoked explicitly, the compiler puts super() as the first statement in the constructor. For example,

```
public A() {
}
```

is equivalent to

```
public A() {
  super();
}
```

```
public A(double d) {
  // some statements
}
```

is equivalent to

```
public A(double d) {
  super();
  // some statements
}
```

Using the Keyword `super`

The keyword `super` refers to the superclass of the class in which `super` appears. This keyword can be used in two ways:

- To call a superclass constructor
- To call a superclass method

CAUTION

You must use the keyword super to call the superclass constructor. Invoking a superclass constructor's name in a subclass causes a syntax error. Java requires that the statement that uses the keyword super appear first in the constructor.

Constructor Chaining

Constructing an instance of a class invokes all the superclasses' constructors along the inheritance chain. This is called *constructor chaining*.

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

1. Start from the main method

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

2. Invoke Faculty constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

3. Invoke Employee's no-arg constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

4. Invoke Employee(String) constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

5. Invoke Person() constructor

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

6. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

7. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

8. Execute println

animation

Trace Execution

```
public class Faculty extends Employee {
  public static void main(String[] args) {
    new Faculty();
  }

  public Faculty() {
    System.out.println("(4) Faculty's no-arg constructor is invoked");
  }
}

class Employee extends Person {
  public Employee() {
    this("(2) Invoke Employee's overloaded constructor");
    System.out.println("(3) Employee's no-arg constructor is invoked");
  }

  public Employee(String s) {
    System.out.println(s);
  }
}

class Person {
  public Person() {
    System.out.println("(1) Person's no-arg constructor is invoked");
  }
}
```

9. Execute println

Example on the Impact of a Superclass without no-arg Constructor

Find out the errors in the program:

```
public class Apple extends Fruit {
}

class Fruit {
  public Fruit(String name) {
    System.out.println("Fruit's constructor is invoked");
  }
}

//Sandwich.java
```

Constructors of Subclass

Every constructor of a subclass must, directly or indirectly, invoke a constructor of its superclass to initialize fields of the superclass. (Subclass cannot access them directly)

Use keyword **super** to invoke constructor of the superclass .

```
public Manager(String n, double s, int
  year, int month, int day)
{
    super(n, s, year, month, day);
    bonus = 0;
}
```

Must be the first line

Constructors of Subclass

- Can call another constructor of subclass.
  - Make sure that constructor of superclass is eventually called.

```
public Manager(String n)
{
   this(n, 0.0, 0, 0, 0);
}
```

Constructor of Subclass

If subclass constructor does not call a superclass constructor explicitly, then superclass uses its default constructor.

```
class FirstFrame extends JFrame
{  public FirstFrame()
   {
      setTitle("FirstFrame");
      setSize(300, 200);
   }
}
```

`super()` implicitly called here

If superclass has no default constructor, compiler error

Constructors of Subclass

Constructors are not inherited.

- Let's say `Employee` has two constructors

```
public Employee(String n, double s, int year,
                int month, int day)
public Employee(String n, double s)
```

- `Manager` has one constructor

```
public Manager(String n, double s, int year,
               int month, int day)
```

```
new Manager("George", 20000, 2001, 7, 20 ); //ok
new Manager("Jin", 25);                     //not ok
```

Declaring a Subclass

A subclass extends properties and methods from the superclass. You can also:

- Add new properties
- Add new methods
- Override the methods of the superclass

Calling Superclass Methods

You could rewrite the printCircle() method in the Circle class as follows:

```
public void printCircle() {
  System.out.println("The circle is created " +
    super.getDateCreated() + " and the radius is " + radius);
}
```

Overriding Methods

Salary computation for managers are different from employees. So, we need to modify the `getSalary`, or provide a new method that overrides `getSalary`

```
public double getSalary( )
  {  double baseSalary = super.getSalary();
     return basesalary + bonus;
   }
```

Cannot replace the last line with

```
salary += bonus;
```

Because **salary** is `private` to `Employee`.

Cannot drop "super.", or else we get an infinite loop

Call method of superclass

Overriding Methods

- An overriding method must have the same **signature** (name and parameter list) as the original method. Otherwise, it is simply a new method:

  – Original Method in `Employee`:

```
public double getSalary( ){…}
public void raiseSalary(double byPercent){…}
```

  – New rather than overriding methods in `Manager`:

```
public void raiseSalary(int byPercent){…}
public void raiseWage(double byPercent){…}
```

Overriding Methods

– An overriding method must have the same return type as the original method:

- The following method definition in `Manager` would lead to compiler error:

```
public int getSalary( ){…}
```

– An overriding method must be at least as visible as the superclass method.

- `private` methods cannot be overridden, but others (public, protected, default-access methods) can.

NOTE

An instance method can be overridden only if it is accessible. Thus a private method cannot be overridden, because it is not accessible outside its own class. If a method defined in a subclass is private in its superclass, the two methods are completely unrelated.

NOTE

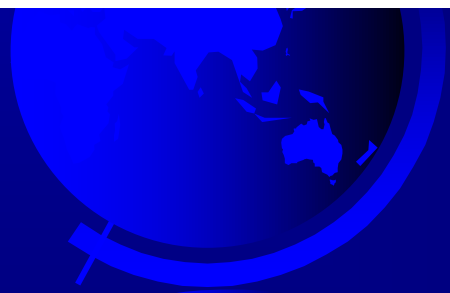
Like an instance method, a static method can be inherited. However, a static method cannot be overridden.

If a static method defined in the superclass is redefined in a subclass, the method defined in the superclass is hidden. (No existence of superclass method.//Wrong

Overriding vs. Overloading

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overrides the method in B
  public void p(double i) {
    System.out.println(i);
  }
}
```

```
public class Test {
  public static void main(String[] args) {
    A a = new A();
    a.p(10);
    a.p(10.0);
  }
}

class B {
  public void p(double i) {
    System.out.println(i * 2);
  }
}

class A extends B {
  // This method overloads the method in B
  public void p(int i) {
    System.out.println(i);
  }
}
```



Additional Methods

```
public void setBonus(double b)
{
   bonus = b;
}
```

Note about Protected Access

- A subclass can access **`protected`** fields and methods of a superclass
- Example: If the **`hireDay`** field of `Employee` is made `protected`, then methods of `Manager` can access it directly.
- However, methods of `Manager` can only access the **`hireDay`** field of **`Manager`** objects, not of other **`Employee`** objects. (See next slide for more explanation)
- Protected fields are rarely used. Protected methods are more common, e.g. **`clone`** of the **`Object`** class (to be discussed later)

```
public class Employee
{   …
    protected Date hireDay;
}

Public class Manager extends Employee
{
      someMethod()
      {
            Employee boss = new Manager();
            boss.hireDay  //ok

            Employee clerk = new
  Employee();
            clerk.hireDay // not ok
      }
}
```

Note about Protected Access

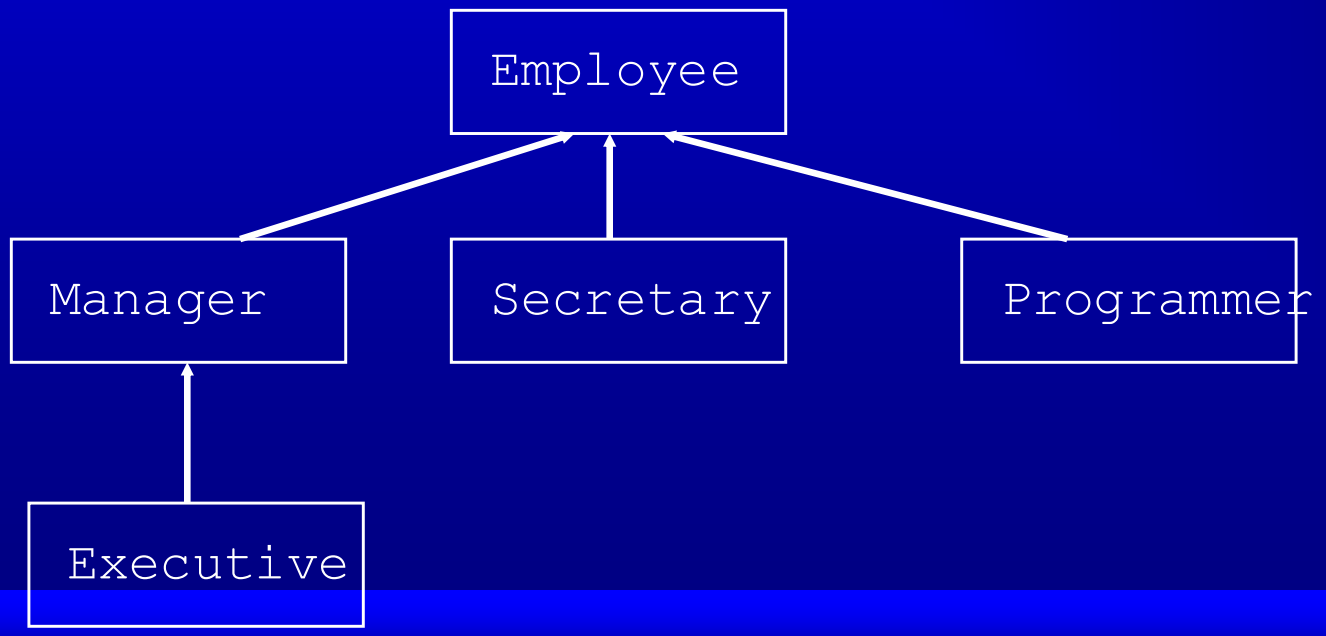
- When to use the `protected` modifier:
  - Best reserved specifically for subclass use.
  - Do not declare anything as protected unless you know that a subclass absolutely needs it.
    - **`clone`** of the **`Object`** class
  - In general, do not declare methods and attributes as protected in the chance that a subclass may need it sometime in the future.
  - If your design does not justify it explicitly, declare everything that is not in the public interface as private.

Note about Inheritance Hierarchies

Can have multiple layers of inheritance:

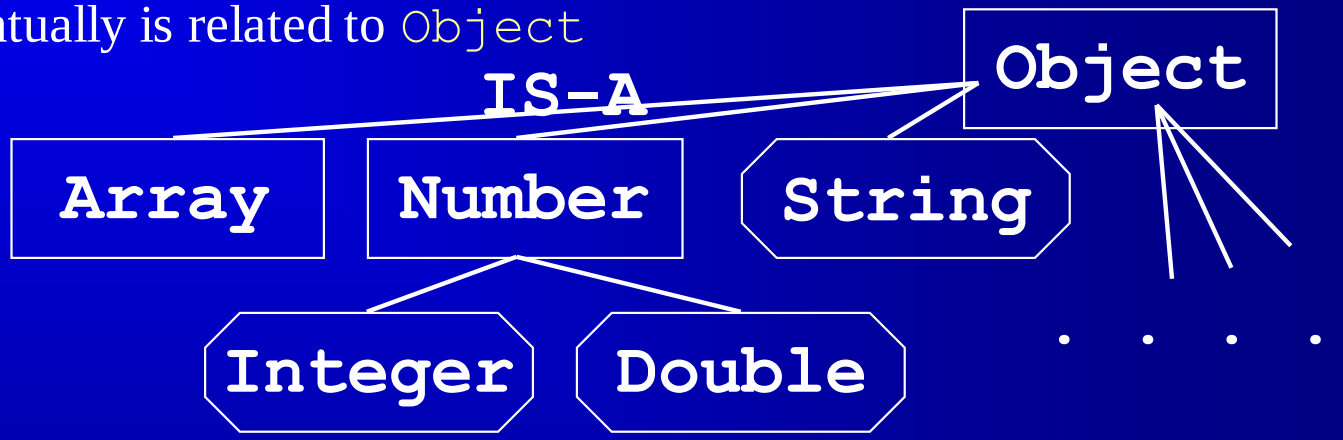
```
class Executive extends Manager { ….}
```

Inheritance hierarchy: inheritance relationships among a collection of classes



The Object Class and Its Methods

- **The** `Object` class (`java.lang.Object`) is the mother of all classes
  - Everything eventually is related to `Object`



  - Never write `class Employee extends Object {…}` because `Object` is taken for granted if no explicitly superclass:

```
class Employee {…}
```

The `Object` class

- Has a small set of methods
  - `boolean equals(Object other);`
    - `x.equals(y);`

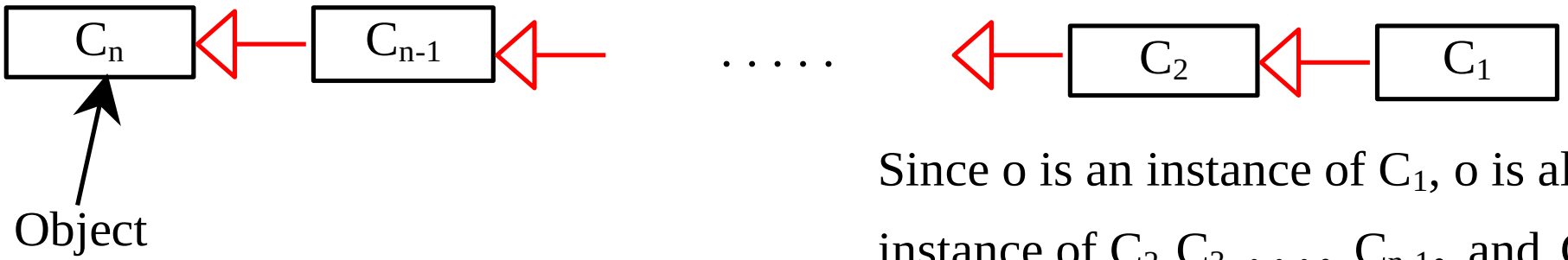
      for any reference values x and y, this method returns

      true if and only if x and y refer to the same object (x==y has the value true).
    - Must be overridden for other equality test, e.g. name, or id

      E.g. Overridden in `String`
  - `String toString();`
    - Returns a string representation of the object
    - `System.out.println(x)` calls `x.toString()`
    - So does `""+x`
    - Override it if you want better format.
    - Useful for debugging support
  - Other methods to be discussed later.

Dynamic Binding

Dynamic binding works as follows: Suppose an object o is an instance of classes C_1, C_2, ..., C_{n-1}, and C_n, where C_1 is a subclass of C_2, C_2 is a subclass of C_3, ..., and C_{n-1} is a subclass of C_n. That is, C_n is the most general class, and C_1 is the most specific class. In Java, C_n is the Object class. If o invokes a method p, the JVM searches the implementation for the method p in C_1, C_2, ..., C_{n-1} and C_n, in this order, until it is found. Once an implementation is found, the search stops and the first-found implementation is invoked.



Since o is an instance of C_1, o is also an instance of C_2, C_3, . . ., C_{n-1}, and C_n

Method Matching vs. Binding

Matching a method signature and binding a method implementation are two issues. The compiler finds a matching method according to parameter type, number of parameters, and order of the parameters at compilation time. A method may be implemented in several subclasses. The Java Virtual Machine dynamically binds the implementation of the method at runtime.

Generic Programming

```
public class PolymorphismDemo {
  public static void main(String[] args) {
    m(new GraduateStudent());
    m(new Student());
    m(new Person());
    m(new Object());
  }

  public static void m(Object x) {
    System.out.println(x.toString());
  }

class GraduateStudent extends Student {
}

class Student extends Person {
  public String toString() {
    return "Student";
  }
}

class Person extends Object {
  public String toString() {
    return "Person";
  }
}
```

Polymorphism allows methods to be used generically for a wide range of object arguments. This is known as generic programming. If a method's parameter type is a superclass (e.g., Object), you may pass an object to this method of any of the parameter's subclasses (e.g., Student or String). When an object (e.g., a Student object or a String object) is used in the method, the particular implementation of the method of the object that is invoked (e.g., toString) is determined dynamically.

Casting Objects

You have already used the casting operator to convert variables of one primitive type to another. *Casting* can also be used to convert an object of one class type to another within an inheritance hierarchy. In the preceding section, the statement

```
m(new Student());
```

assigns the object new Student() to a parameter of the Object type. This statement is equivalent to:

```
Object o = new Student(); // Implicit casting
m(o);
```

The statement Object o = new Student(), known as implicit casting, is legal because an instance of Student is automatically an instance of Object.

Why Casting Is Necessary?

Suppose you want to assign the object reference o to a variable of the Student type using the following statement:

```
Student b = o;
```

A compilation error would occur. Why does the statement **Object o = new Student()** work and the statement **Student b = o** doesn't? This is because a Student object is always an instance of Object, but an Object is not necessarily an instance of Student. Even though you can see that o is really a Student object, the compiler is not so clever to know it. To tell the compiler that o is a Student object, use an explicit casting. The syntax is similar to the one used for casting among primitive data types. Enclose the target object type in parentheses and place it before the object to be cast, as follows:

```
Student b = (Student)o; // Explicit casting
```

Casting from Superclass to Subclass

Explicit casting must be used when casting an object from a superclass to a subclass. This type of casting may not always succeed.

```
Apple x = (Apple)fruit;

Orange x = (Orange)fruit;
```

Using Subclasses

- Class compatibility:
  - An object of subclass is automatically an object of superclass, but not vice versa.
    - `Employee harry = new Employee();`
    - `Employee jack = new Manager();`
- Polymorphism:
  - Object variable an refer to multiple actual types
- Dynamic binding
  - Java's ability to call the appropriate method depending on actual type of object
    - harry.getSalary();
    - jack.getSalary();

Class Compatibility

Object of a subclass can be used in place of an object of a superclass

```
Manager harry = new Manager(...);
Employee staff = harry;
Employee staff1 = new Manager(...);
```

`harry` automatically cast into an `Employee`, widening casting.

Why does `staff.getSalary()` work correctly?

- Employee has method `getSalary`. No compiling error.
- Correct method found at run time via dynamic binding

Class Compatibility

The opposite is not true

```
Employee harry = new Employee(...);
Manager staff = harry;// compiler error
Manager staff1 = new Employee(...); //
compiler error
```

The `equals` Method

The `equals()` method compares the contents of two objects. The default implementation of the equals method in the Object class is as follows:

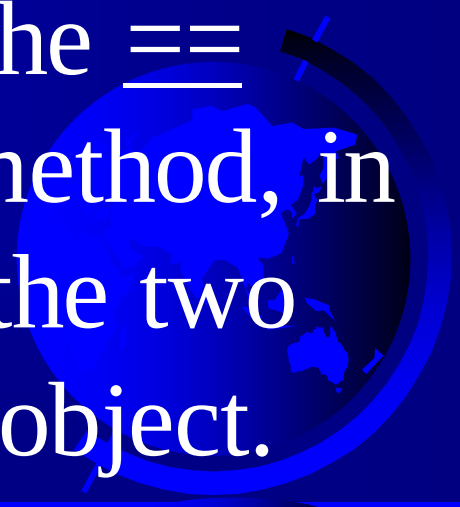
```
public boolean equals(Object obj) {
  return (this == obj);
}
```

For example, the equals method is overridden in the Circle class.

```
public boolean equals(Object o) {
  if (o instanceof Circle) {
    return radius == ((Circle)o).radius;
  }
  else
    return false;
}
```

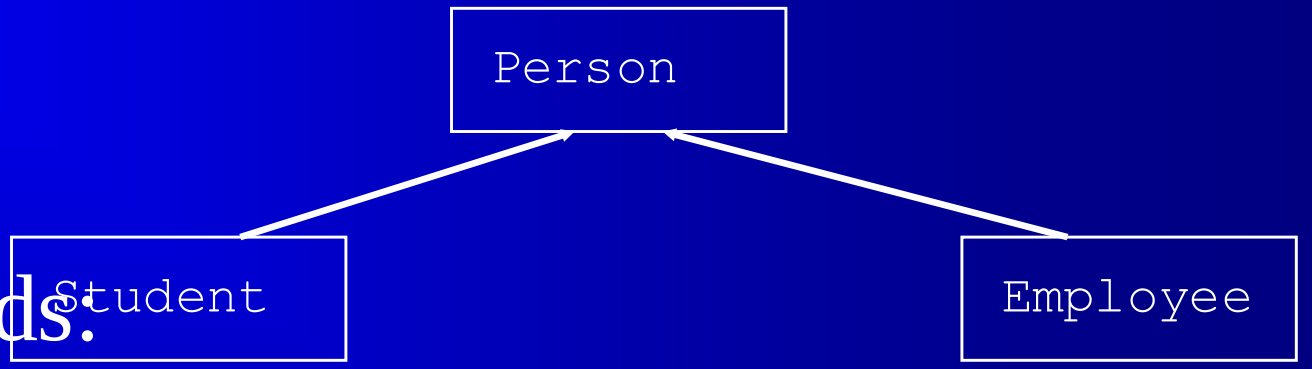
NOTE

The == comparison operator is used for comparing two primitive data type values or for determining whether two objects have the same references. The equals method is intended to test whether two objects have the same contents, provided that the method is modified in the defining class of the objects. The == operator is stronger than the equals method, in that the == operator checks whether the two reference variables refer to the same object.



Abstract Classes

- Consider two classes: `Employee` and `Student`



- Common methods:
  - `getName`
  - `getDescription`: returns
    - `Employee`: "an employee with salary $50,000"
    - `Student`: "a student majoring in Computer Science"

- To write better code, introduce a superclass `Person` with those two methods

```
class Person
{   public Person(String n) {    name
  = n;}
    public String getName()
    { return name;}

    public String getDescription();
   // but how to write this?

   private String name;
}
```

- The `Person` class knows nothing about the person except for the name. We don't know how to implement the method in the `Person` class although we know it must be there.

- Solution: leave the **`getDescription`** method abstract
  - Hence leave the `Person` class abstract

```
abstract class Person
{   public Person(String n)
    {    name = n;}

    public abstract String
  getDescription();

    public String getName()
    { return name;}
    private String name;
}
```

- An abstract method is a method that
  - Cannot be specified in the current class (C++: pure virtual function).
  - Must be implemented in non-abstract subclasses.

- An abstract class is a class that may contain one or more abstract methods

- Notes:
  - An abstract class does not necessarily have abstract method
  - Subclass of a non-abstract class can be abstract.

- Cannot create objects of an abstract class:

```
New Person("Micky Mouse")   // illegal
```

- An abstract class must be extended before use.

```
class Student extends Person
{  public Student(String n, String m)
   {  super(n);  major = m;}

   public String getDescription()
   { return "a student majoring in " + major; }
   private String major;
}
```

```
class Employee extends Person
{   …
    public String getDescription()
    {
        NumberFormat formatter
            = NumberFormat.getCurrencyInstance();
        return "an employee with a salary of "
            + formatter.format(salary);
    }
    …
    private double salary;
}
```

```
Person[] people = new Person[2];
people[0]= new Employee("Harry
  Hacker", 50000, 1989,10,1);
people[1]= new Student("Maria
  Morris", "computer science");

for (int i = 0; i <
  people.length; i++)
{    Person p = people[i];

  System.out.println(p.getName() +
  ", "
                +
  p.getDescription());
}   //PersonTest.java
```

Final Methods and Classes

- Final method:
  - Declared with keyword **final**
  - Cannot be overridden in subclasses

```
class Employee
{    …
     public final String getName()  {…}
}
```

Final Methods and Classes

- Final class
  - Declared with keyword `final`
  - Cannot be sub-classed. Opposite of abstract class.
  - All methods are `final`.

```
final class Executive extends Manager
{ …. }
```

- Reasons to use `final` methods (and `final` classes):
  - Efficiency:
    - Compiler put final method in line: `e.getName()` replaced by `e.name`.
    - No function call.
    - No dynamic binding.
  - Safety:
    - Other programmers who extend your class cannot redefine a final method.

The `final` Modifier

- The `final` class cannot be extended:

```
final class Math {
  ...
 }
```

- The `final` variable is a constant:

```
final static double PI = 3.14159;
```

- The `final` method cannot be overridden by its subclasses.

Summary of Modifiers

- Class Modifiers
  - `public`: Visible from other packages
  - default (no modifier): Visible in package
  - `final`: No subclasses
  - `abstract`: No instances, only subclasses
- Field modifiers
  - `public`: visible anywhere
  - `protected`: visible in package and subclasses
  - default (no modifier): visible in package
  - `private`: visible only inside class
  - `final`: Constant

Summary of Modifiers

- Method Modifiers
  - `final`: No overriding
  - `static`: Class method
  - `abstract`: Implemented in subclass
  - `native`: Implemented in C
  - `private, public, protected, default`: Like variables